Quicksilver: A Proxy App for the Monte Carlo Transport Code Mercury

Workshop on Representative Applications

David Richards, Ryan Bleile, Patrick Brantley,
Shawn Dawson, Scott McKinley, Matthew O'Brien

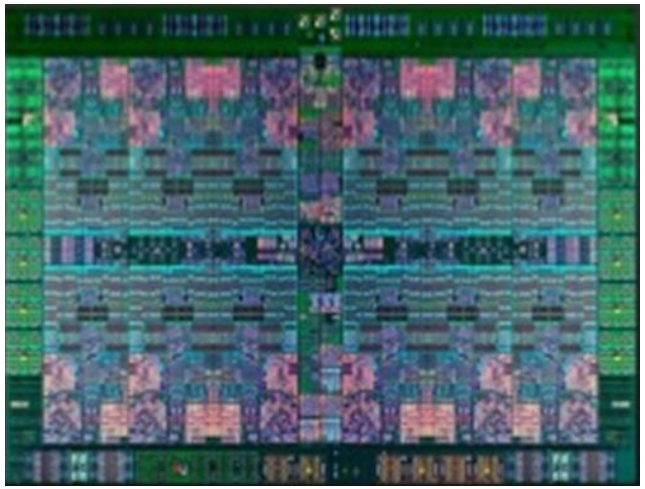
September 5, 2017

LLNL-PRES-743737

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. Lawrence Livermore National Security, LLC

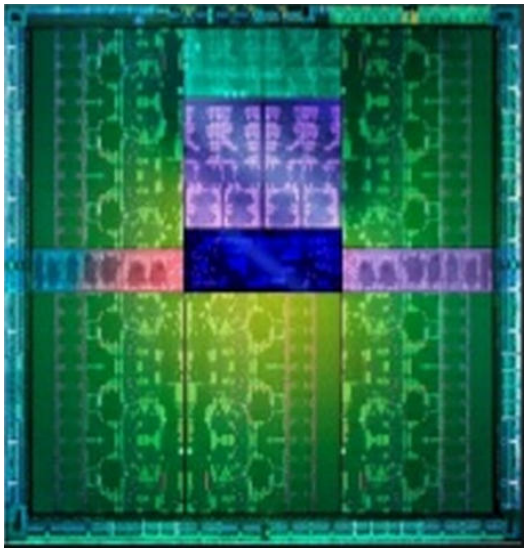
Lawrence Livermore National Laboratory

Sierra will include IBM Power9 CPUs and Nvidia Volta GPUs



IBM Power 9 CPU
<10% Sierra Flops

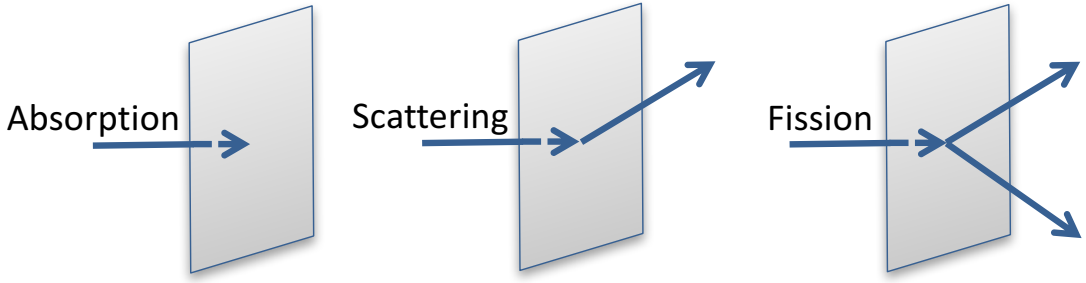
~4500 nodes
Each node has:
2 Power9
4 Volta GPU



Nvidia Volta GPU
> 90% Sierra Flops
16 GB HBM/GPU

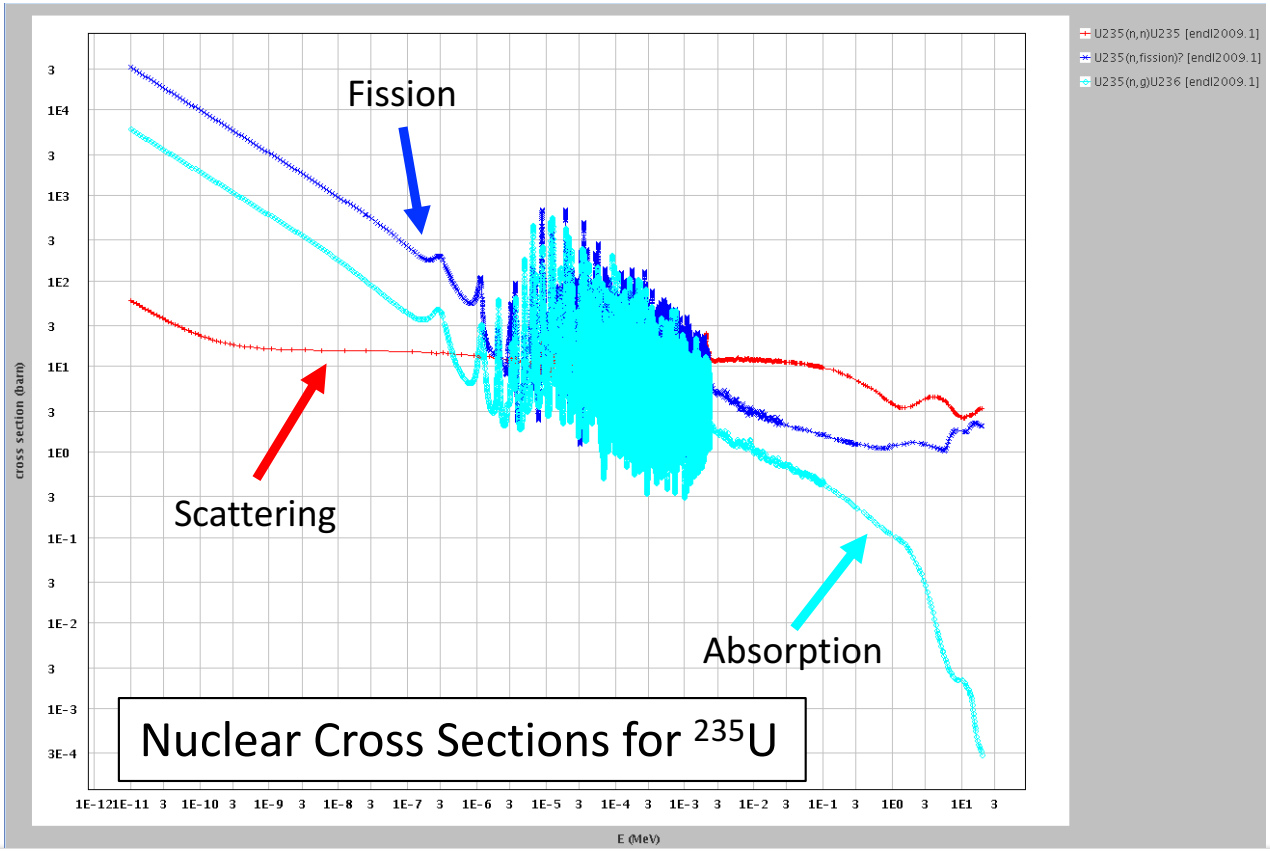
Mercury solves particle transport problems using the Monte Carlo Method

- Particles interact with matter by a variety of "reactions"



- The probability of each reaction and its outcomes are captured in experimentally measured "cross sections" (Latency bound table lookups)
- Follows many particles (millions or more) and uses random numbers to sample the probability distributions (Very branchy, divergent code)
- Particles contribute to diagnostic "tallies" (Potential data races)
- The result is a statistically correct representation of the physical system

Nuclear cross sections are not “nice” functions



Creating Quicksilver required modeling choices

Proxy apps are models for one or more aspects of their parents

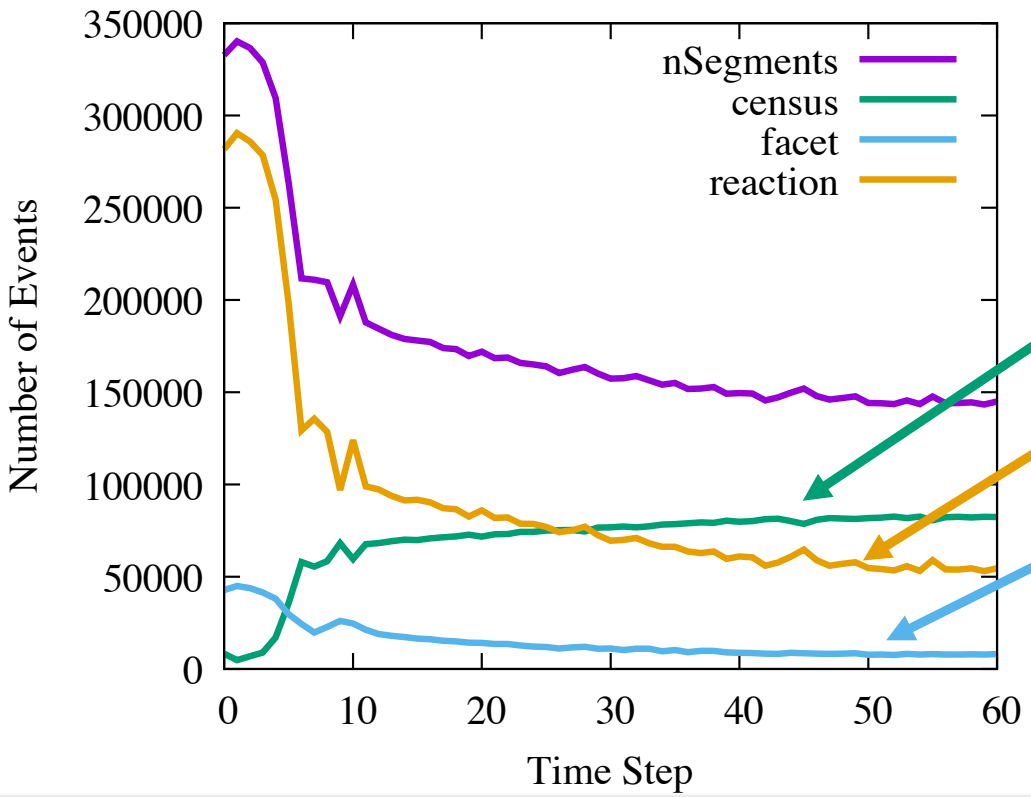
- Three specific uses:
  - A nimble prototype code for testing design or refactoring options for Mercury
  - An open source vehicle for co-design with outside partners
  - A benchmark code to replace our previous Monte Carlo benchmark code
- Overall goal was to approximate the overall application performance of Mercury
  - Control flow is dominated by branching due to the random sampling of reactions.
  - Memory access patterns associated with reading cross section tables tend to be latency-bound, small memory loads that are difficult or impossible to cache or coalesce.
  - Domain decomposition and internode communication to handle large problems.
- Major data structures intentionally similar to Mercury
- Flexible inputs to represent multiple common use modes

It is essential to to identify the key features of the parent app the proxy is intended to represent and include faithful models of those features

Quicksilver omits many Mercury features, but keeps enough to represent critical computational patterns

	Mercury	Quicksilver
Cross Sections	Continuous energy & multi-group	Multi-group (synthetic data)
Mesh/Geometry	Multiple types & solid geometry	3D polyhedral only
Reactions	10-40	3 (uses replication)
Reaction Physics	Physically based	Simplified (isotropic, etc.)
Tallies	Many built-in & user defined	Balance & scalar flux
Sources & population ctrl	Realistic with variance reduction	Simplified
Load balancing	Sophisticated application specific	Trivial particle-count based
Input specification	Python scripting interface	Flexible problem setup (YAML)
MPI/OpenMP	Yes/Yes	Yes/Yes

Is Quicksilver a good representation of Mercury?



Balance tallies count each kind of event.
Data is for 100,000 particles.

Particles reaching census increases

Number of reactions drops sharply

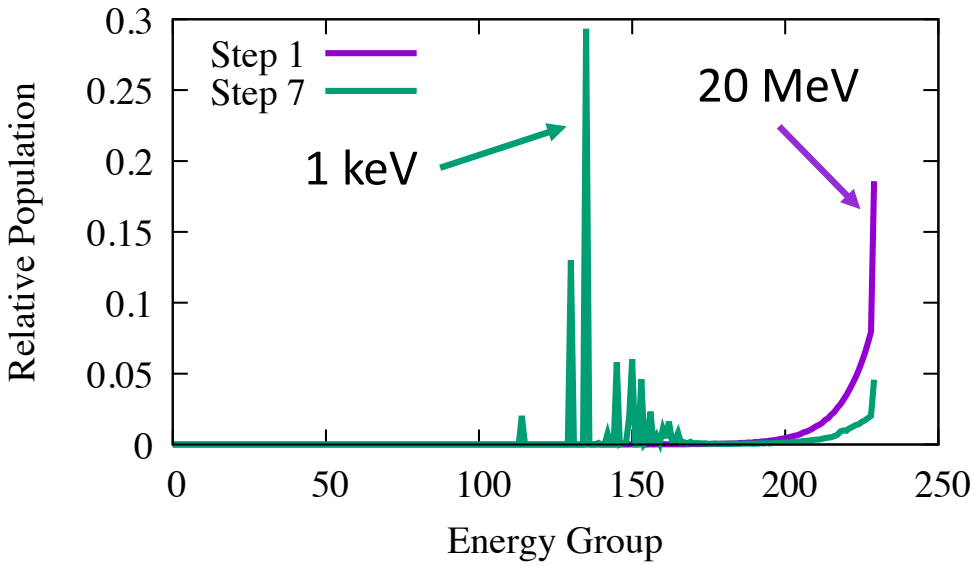
Facet crossings nearly vanish

Big changes during first 10 time steps.
Not the behavior we expected

The particle energy spectrum has an unexplained shift.

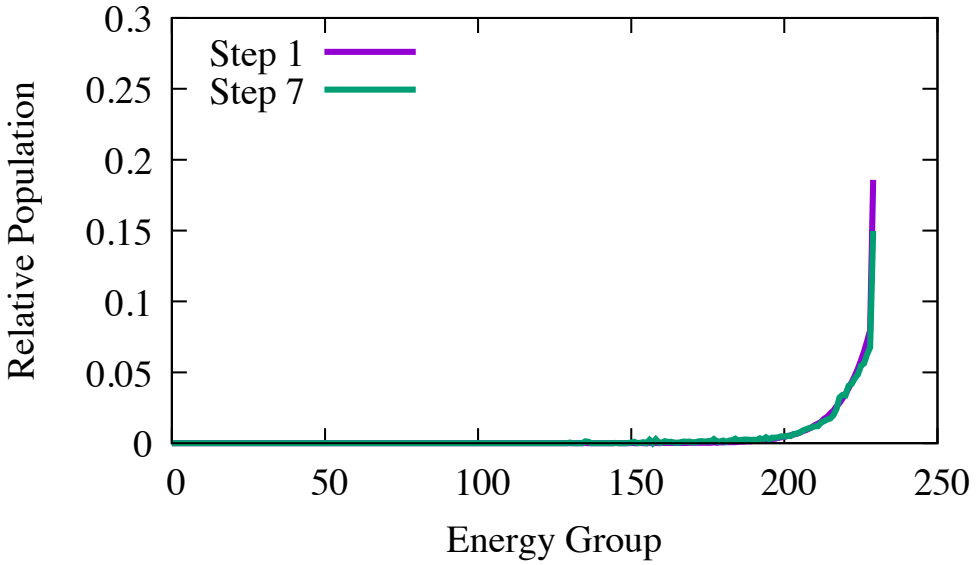
Caused by unintended consequences of simplified physics & sourcing.

- Sourced particles have random energy drawn from uniform distribution
- Simplified sourcing rules create 10% of target simulation particles each step
- Population control splits or kills particles to achieve target number
  - Adjusts particle statistical weight
- In just a few time steps, population control magnifies any particles that survive to census
  - Rare, low energy particles less likely to be absorbed



Energy groups are logarithmic. Particles in group 135 are about 100x lower velocity than group 230

Deleting low-weight particles solves the problem



There is no such capability in the parent application, but it makes Quicksilver a better model for Mercury

Quicksilver and Mercury are hostile to the typical GPU fine-grained threading approach

- loop over cycles (time steps)
  - cycle_init
    - source in new particles
    - population control
  - cycle_tracking
    - loop over particles
      - until census
        - find distance to census (end of time step)
        - find distance to material boundary (mesh facet)
        - find distance to collision (reaction)
        - select reaction and update particle
  - cycle_finalize

This is 1000s (or 10,000s) of lines of code

Majority of cross section look ups are in here

Quicksilver and Mercury are hostile to the typical GPU fine-grained threading approach

- loop over cycles (time steps)
  - cycle_init
    - source in new particles
    - population control
  - cycle_tracking
    - loop over particles
      - until census
        - find distance to census (end of time step)
        - find distance to material boundary (mesh facet)
        - find distance to collision (reaction)
        - select reaction and update particle
  - cycle_finalize

"Fat" threading strategy:
- Each thread gets its own "vault" of particles
- Tally and buffer data structures are replicated to avoid races
- Works great on CPU platforms!

This is 1000s (or 10,000s) of lines of code

Majority of cross section look ups are in here

How do you write this code for GPUs?

Quicksilver and Mercury are hostile to the typical GPU fine-grained threading approach

- loop over cycles (time steps)
  - cycle_init
    - source in new particles
    - population control
  - cycle_tracking
    - loop over particles
      - until census
        - find distance to census (end of time step)
        - find distance to material boundary (mesh facet)
        - find distance to collision (reaction)
        - select reaction and update particle
  - cycle_finalize

"Fat" threading strategy:
- Each thread gets its own "vault" of particles
- Tally and buffer data structures are replicated to avoid races
- Works great on CPU platforms!

This is 1000s (or 10,000s) of lines of code

Majority of cross section look ups are in here

Make this a kernel!

Can this "Big-Kernel" approach possibly perform well?

To build a big kernel, Mercury's threading model must change

Fat thread

- Dozens of active threads
- Separate collection of particles for each thread
- Data races managed with replication
- MPI tightly integrated in tracking loop
- Works well on CPUs

Thin thread

- Thousands of active threads
- All threads share a common collection of particles
- Data races managed with atomics
- No MPI in tracking loop
- Works on GPUs and CPUs

To test big-kernel we wrote a proxy app for our proxy app

- Quicksilver was more complicated than we wanted to port GPU
  - MPI, variable particle count, etc.
- Quicksilver_lite is even more approximate than Quicksilver
  - Zero-D mesh, very simplified physics
- Quicksilver_lite maintains features most likely to impair GPU performance
  - Random table look-ups
  - Call stack depth in nuclear data look-ups
  - Branchy control flow and divergence

QS_lite run times (lower is better)

	Initialize	Compute
P8 CPU (10 threads)	0.27 sec	1.25 sec
P8 CPU (40 threads)	0.45 sec	0.72 sec
P-100 GPU	0.26 sec	0.45 sec

QS_lite provided our first evidence that the big-kernel approach might actually work

Mercury is employed for a very wide variety of problems

No single sample problem will represent all use cases

- By changing problem inputs we can adjust:
  - Fraction of particles that reach census
  - Ratio of facet crossings to reactions
  - Relative probabilities of different reaction types

	Fat (CPU) seg/sec	Thin (GPU) seg/sec
Reaction Dominated	8.52e+06	1.15e+07
Balanced	1.35e+07	7.50+e06
Facet Dominated	2.24e+07	2.90+e07

Higher is better

GPUs and CPUs are similar, but with difference performance sensitivities.
GPUs are slowest in balanced case. Perhaps due to highest divergence?

Performance comparison to Mercury

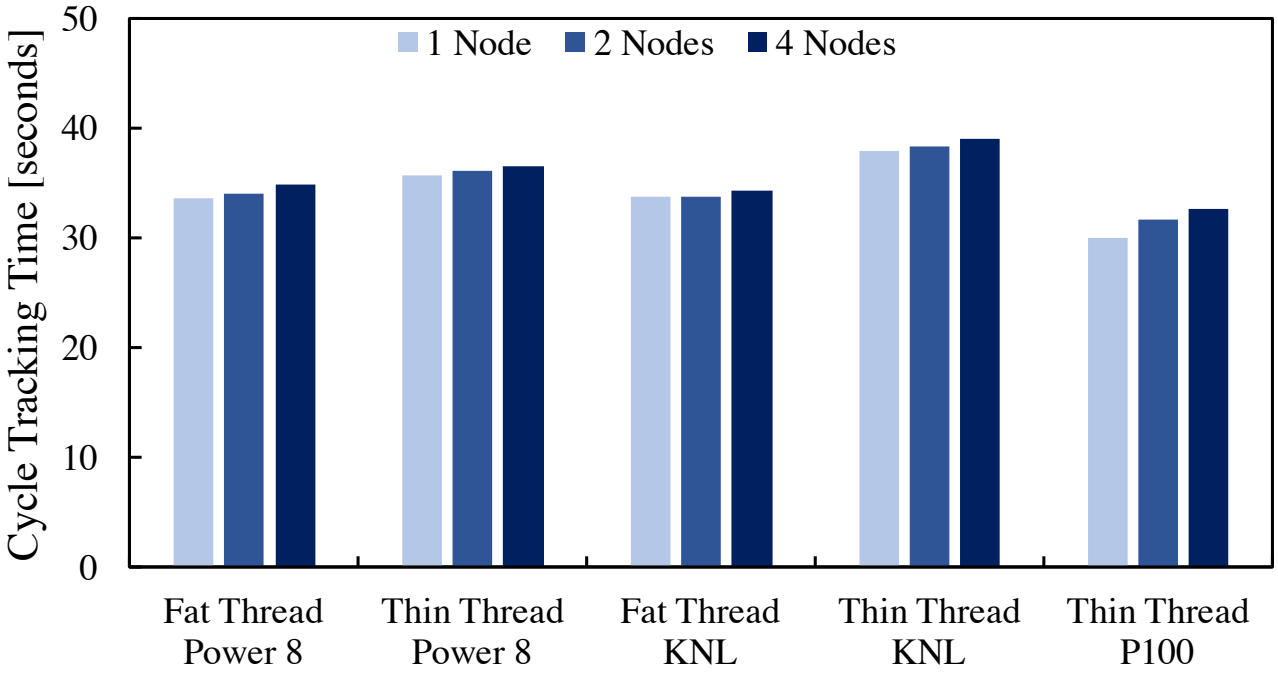
- Different code capabilities make apples-to-apples comparison impossible
- Mercury test problem is a critical sphere in water
  - Tuned mesh size and time step to obtain same tally ratios as a similar Quicksilver problem

CPU Performance	Mercury (seg/sec)	Quicksilver (seg/sec)
Reaction Dominated	1.62e+06	8.52e+06
Balanced	2.66e+06	1.37e+07
Facet Dominated	2.97e+06	2.24e+07

Higher is better

Quicksilver is roughly 10x faster than Mercury on CPUs, but captures same trends. Performance difference likely somewhat due to Mercury's better physics.

Will big kernel work?



Cycle tracking time for a reaction dominated problem (lower is better)

In spite of adverse algorithmic characteristics, we are hopeful that Mercury will perform equally well on GPUs as CPUs. A potential 3-5x speedup compared to CPUs only

Conclusion and future work

- We used Quicksilver to test design strategies for Monte Carlo Transport on GPUs
  - So far, a big kernel approach appears to be viable
- Effort is shifting from prototyping with Quicksilver to refactoring Mercury.
  - Design role is mostly done
- Modifications are planned to make Quicksilver more representative for photons
  - This will make Quicksilver a more flexible procurement benchmark
  - More on proxies vs benchmarks in Thursday keynote talk

Lawrence Livermore
National Laboratory